HyGN: Hybrid Graph Engine for NUMA

Tanuj Aasawat[1], Tahsin Reza[2,3], Kazuki Yoshizoe[1], Matei Ripeanu[3]
[1]RIKEN Center for Advanced Intelligence Project, [2]Lawrence Livermore National Laboratory, [3]University of British Columbia
{tanujkr.aasawat, kazuki.yoshizoe}@riken.jp, {treza, matei}@ece.ubc.ca

***Abstract*—Modern shared-memory platforms embrace the Non-uniform Memory Access (NUMA) architecture - they have physically distributed, yet cache-coherent shared-memory. This paper explores the feasibility of a shared-memory graph processing engine for NUMA platforms inspired by designs that target zero-sharing platforms. This work exploits the characteristics of two processing modes, *synchronous* and *asynchronous*, in the context of the shared-memory NUMA platform. Depending on the algorithm, phase of execution, and graph topology, synchronous and asynchronous modes hold unique advantages over one another. We then explore a hybrid solution that combines synchronous and asynchronous processing within the same graph computation task and harness optimizations therein. An extensive evaluation using graphs with billions of edges and empirical comparisons with several state-of-the-art solutions demonstrate the performance advantages of our design.**

I. Introduction

Depending on the supporting platform, high-performance graph processing solutions can be divided as: *(i)* scale-out solutions that target a shared-nothing platform (e.g., a compute cluster); and *(ii)* single-node solutions that target a shared-memory platform. While a scale-out distributed solution (e.g., GraphLab [1] and HavoqGT [2]) is indeed needed to process massive graphs, today's shared-memory platforms equipped with up to terabytes of main memory and hundreds of hardware threads, are often a better choice for medium size workloads since they offer higher performance for a fixed dollar or watt budget [3]–[6].

Modern multiprocessor (i.e., multi-socket) platforms embrace the Non-uniform Memory Access (NUMA) architecture. This architecture distributes memory to each processor/socket to reduce contention on the memory bus; and enables shared-memory platforms with massive main memory and a large number of processors. However, the cost of accessing memory is non-uniform: accessing a memory location belonging to another socket (a so-called *remote* memory access) is slower than accessing a memory location belonging to the same socket where the thread generating the memory access request runs (a so-called *local* memory access). Remote memory accesses are 2–7.5× more expensive than that of the local memory ones [7]. Local and remote memory access performance also varies with access type (e.g., read vs. write) and patterns (sequential vs. random). The remote-random access can be twice as expensive as the local-random access, and over 5× more expensive than the local-sequential access. Therefore, careful data placement and memory access pattern optimizations are needed to maximize application performance.

TABLE I: Relative speedup in runtime on two and four sockets, over a single socket (where all memory accesses are local), for two high-performance graph processing systems (Galois and Totem), three algorithms, and on two different workloads (synthetic - RMAT-29, and real-world - Twitter). See §III for experiment details.

#Sockets	Algorithm	Galois (Twitter)	Totem (RMAT-29)
2	BFS	0.98×	1.54×
	SSSP	1.49×	1.35×
	PageRank	1.32×	1.37×
4	BFS	1.72×	1.97×
	SSSP	1.45×	1.78×
	PageRank	1.67×	2.41×

Why NUMA Awareness is Crucial? Unfortunately, most existing graph processing systems are unable to harness the full performance potential of modern NUMA systems, and as a consequence, they do not scale well with the increasing number of NUMA nodes (i.e., processors/sockets). Table I highlights the limited scalability of two well-known frameworks Galois [4] and Totem [3], for three graph algorithms (see §III for experiment details). These solutions do not scale well with the number of NUMA nodes; primarily due to the fact that they are NUMA-oblivious.

One can view a NUMA system as resembling a *high bandwidth* and *low latency*, shared-nothing distributed platform, but with the flexibility of accessing both local and remote memory in the same byte addressable manner; hence, the message passing overhead of remote accesses is significantly lower than for protocols used by shared-nothing platforms. This view presents the opportunity to transfer some of the wisdom of distributed graph processing platforms to develop a shared-memory graph processing solution optimized for NUMA systems.

We present **HyGN** (short for Hybrid Graph processing engine for NUMA) - a graph processing engine optimized for NUMA platforms. Similar to scale-out solutions we partition the graph and bind each partition to a NUMA node to maximize locality. Each partition is processed by parallel threads available in the NUMA node. HyGN supports two key processing modes: *synchronous* - the compute phase only operates on the process/node local data, and an explicit communication phase synchronizes the required distributed algorithm states; *asynchronous* - communication is overlapped with compute, and no explicit communication phase is required. Additionally, the engine supports a *hybrid* computation mode which combines sync and async processing within the same graph computation task execution.

IEEE BigData20, December 10-13, 2020
978-1-7281-6251-5/20/$31.00 ©2020 IEEE

TABLE II: Graph processing frameworks and their design characteristics. Following distributed frameworks are NUMA-oblivious. GraphLab also supports Async, however, the Sync mode offers vastly superior performance [10].

	Framework	Processing mode	NUMA-aware	Distr. Comm. and/or Parallelism	Remote Traffic Aggregation
Distributed	GraphLab	Sync	No	MPI+OpenMP	Yes
	HavoqGT	Async	No	MPI	N/A
	GraphMat	Sync	No	MPI+OpenMP	Yes
	Gluon-Async	Bulk-Async	No	LCI [11]+Pthread	Yes
	PowerSwitch	Hybrid (Sync+Async)	No	MPI+OpenMP	Yes
Single node	Galois	-	No	Pthread	N/A
	Ligra	-	No	Cilk	Yes
	Totem	-	No	OpenMP	N/A
	Polymer	Sync	Yes	Pthread	Yes
	GraphGrind	Sync	Yes	Cilk	Yes
	HyGN	Sync, Async, Hybrid (Sync+Async)	Yes	OpenMP	Yes

HyGN's design draws inspiration from shared-nothing scale-out designs, however, it aims to take advantage of the shared-memory (in Table IV we directly compare with two scale-out solutions, out-of-the-box). Past contributions Polymer [7] and GraphGrind [8] only explored synchronous distributed processing for NUMA platforms. PowerSwitch [9] enables synchronous and asynchronous hybrid processing, however, was explored in the context of scale-out distributed processing; and compared to ours, it adopts a different strategy to select a processing mode (explained in §II-D). Table II presents the key design characteristics of a number of well-known shared-memory and distributed graph frameworks and how they compare with HyGN.

This paper makes the following contributions:

(i) We present a detailed study that highlights the advantages and shortcomings of synchronous and asynchronous processing modes with respect to different algorithms and graph topologies; and a thorough design exploration that justifies the need for Sync+Async hybrid processing on the NUMA platform (§II-C).

(ii) Based on these insights we present the design and implementation of HyGN: a *fast* and *memory efficient* graph processing engine. We evaluate HyGN using four well-known graph algorithms (covering a wide spectrum of memory access patterns and communication behaviours) namely, Breadth-first search (BFS) (two variants) [12], Single Source Shortest Path (SSSP - Bellman-Ford algorithm), and PageRank (PR). Through evaluation on a four socket NUMA machine, and using six real-world and four synthetic graphs - with up to 85 billion edges, we demonstrate the performance and scalability of our solution (§III). Across these algorithms our solution shows on average 3× (up to 3.9×) scalability on four sockets (§III-C) and achieves up to 73 Billion TEPS (Traversed Edges Per Second) throughput (§III-A) for BFS.

(iii) We empirically compare our solution with five related systems: the NUMA-aware Polymer [7]; state-of-the-art NUMA-oblivious solutions Galois and Totem; and distributed frameworks PowerSwitch and HavoqGT. We demonstrate up to 45× speedup (on average 7.7×) over Galois; and up to 14× speedup (on average 3.9×) over NUMA-aware Polymer (§III-B). Additionally, our system is 5× more memory efficient than the most comparable system, Polymer.

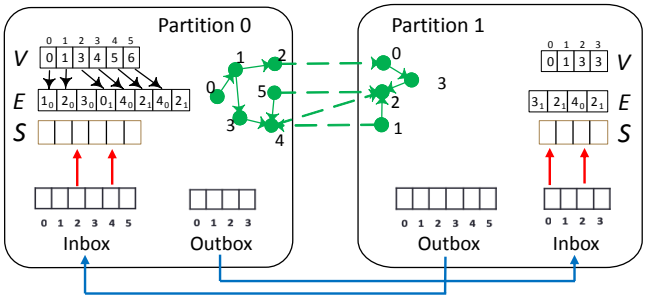


Fig. 1: High-level illustration of graph partitions in HyGN. V (vertex) and E (edge) are the arrays to represent each partition of the graph in the CSR format. S is the state buffer for local vertices of a partition. Each outbox is the state buffer for remote neighbors within the same partition. The figure shows state buffers (i.e., the inbox and outbox) required for the synchronous communication design; and the communication paths for explicit memory copy from the out- to in-box. This design enables, in synchronous mode, a partition to perform lock-free parallel memory copy to all remote partitions. Updates from remote inbox(es) are then applied to the local state buffer of the destination partition. A pair of in- and out-boxes are of the same size, determined at graph partitioning time. In the asynchronous design, remote threads directly update a remote vertex's state in S (i.e., the in- and out- box state buffers are not used).

II. HyGN: System Design and Implementation

This section presents the design and implementation of HyGN. The system is comprised of three functional elements: *(i)* a graph partitioning module, *(ii)* a parallel compute engine, and *(iii)* a communication substrate; below, we describe them.

A. Graph Partitioning

The graph is split into the same number of partitions as the available NUMA domains of the target platform. We adopt a light-weight partitioning approach[1]: First, the graph vertices are sorted by their neighbor degree (past work has demonstrated this improves cache locality [3], thus performance); then the graph vertices are distributed among the partitions in a round-robin manner; resulting in each partition to have approximately equal share of graph vertices and edges. Within each partition, the subgraph is maintained in the compact Compressed Sparse Row (CSR) data structure. Each vertex is assigned an additional vertex ID, local to the partition it belongs to; the higher order bits in the vertex ID also encode the vertex's 'home' partition ID. The neighbors (i.e., the target vertex of an edge) of each vertex is identified as *local* or *remote*. A neighbor is local if its home partition ID is the same as that of the vertex, otherwise the neighbor is marked remote. Fig. 1 shows an example using two partitions.

B. Parallel Compute Engine

HyGN's graph processing engine embraces an iterative design similar to the popular Bulk Synchronous Parallel (BSP)

[1]Other partitioning techniques that may offer further load balancing are possible within our design (the partitioning is likely much more expensive).

model: in each *superstep*, the graph partitions work in parallel to process the graph vertices. In the communication phase (in Sync mode only), the distributed algorithm states are synchronized via inter-partition communication.

In the *compute phase* of a superstep, for each partition, the system iterates over local vertices and process them. At the individual partition-level, we adopt a *vertex-centric* processing model: a fixed pool of threads (local to a NUMA domain) iteratively process the vertices. For a graph traversal algorithm, the processing task is essentially a *neighborhood computation*: either a *scatter* operation - a vertex updates the algorithm state of its neighbors, e.g., the *distance* in the SSSP algorithm; or a *gather* operation - a vertex accumulates the algorithm state of the neighbors, for instance, to compute its own state, e.g., the *rank* in the PageRank algorithm.

At application launch time, for each graph partition, we allocate the graph topology and algorithm specific data structures on the respective NUMA node using the *libnuma* library. We use nested parallelism supported in OpenMP for parallel processing: first, each NUMA node creates a single thread that performs the above discussed memory allocations for each graph partition. Then, each of these threads spawns child threads (equal to the number of cores/hyper-threads available on a NUMA node) for algorithm processing. The *main* thread launches a superstep; parallel threads, one per NUMA node, invoke the compute kernel on all partitions; each partition spawns child threads (equal to the number of cores/hyper-threads) for algorithm processing.

C. Distributed Processing Modes and Inter-partition Comm.

This section first presents the design of the infrastructure enabling synchronous and asynchronous processing, then evaluates the performance of each technique, and highlights their respective advantages and limitations. Informed by this study, the following section presents the design of hybrid mode for iterative graph processing on NUMA platforms.

Synchronous (Sync) Mode. Here, we embrace a BSP (Bulk Synchronous Parallel) model and we consider NUMA as a shared-nothing platform where each socket represents an independent node. Each superstep has a computation and a communication phase. In the computation phase, memory accesses are limited within a partition's local NUMA domain and updates to remote neighbors' states are stored locally.

During computation, each partition updates its local state buffer, S, for local neighbors. For each remote neighbor, its state is locally updated in the corresponding outbox. In the communication phase, each outbox is copied to the corresponding inbox of the remote partition, and then from the inbox to the local state buffer S (Fig. 1). All partitions perform this memory copy between NUMA domains in parallel.

The key advantages of this design are: *(i)* Zero remote memory accesses during the compute phase. *(ii)* All remote memory writes (outbox to inbox copy) are sequential. *(iii)* This approach enables *message aggregation* - a partition maintains only one state per remote vertex (in the corresponding outbox), irrespective of the number of incoming edges the remote vertex has. This has the potential of significantly decreasing the volume of inter-partition traffic, therefore, remote communication; especially in the presence of high-degree vertices that are prevalent in scale-free graphs.

Asynchronous (Async) Mode. Since, NUMA platforms are shared-memory systems, there are opportunities for design optimizations over solutions targeting shared-nothing platforms, especially with respect to remote communications. A NUMA node can directly access data stored in a remote NUMA domain simply by obtaining a pointer to the corresponding address (enabled by the virtual memory management subsystem); no explicit message passing over an expensive protocol (e.g., MPI) is required.

Unlike Sync, Async does not have an explicit communication step for synchronizing distributed algorithm states; instead, during computation, a vertex directly updates the state of a remote neighbor in the state buffer residing in its home partition (in a remote NUMA domain). This design does not require the outbox or the inbox; it performs remote access during computation, however, they are *overlapped by computation* (by the other threads), which helps to hide the overhead of remote accesses. One design limitation of Async is, unlike the Sync mode, it does not support the message aggregation mechanism (described earlier); therefore, it will result in higher inter-partition traffic compared to the Sync mode. Furthermore, the remote memory access are random (unlike in Sync mode where they are sequential).

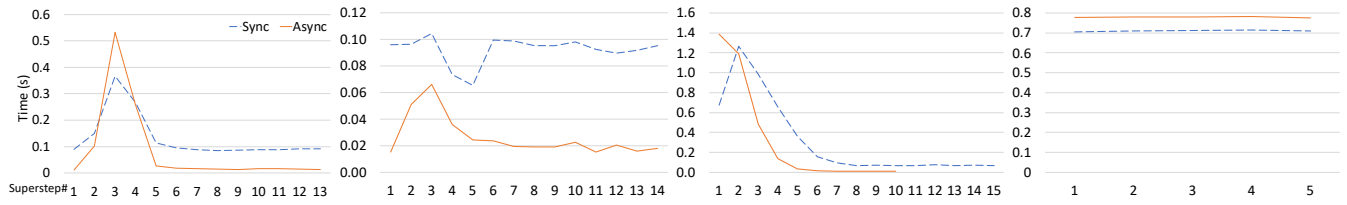
Comparing Sync and Async Performance. Since Sync aggregates messages for remote vertices, it performs faster than Async for PageRank (where the frontier is always large), as shown in Fig. 4; as Async suffers from doing many expensive remote random accesses. For BFS-DO, where few updates happen in a superstep, Async benefits from overlapping computation with communication and updating the visited remote neighbor directly in their respective partitions. Here, Async is 2.8× faster than Sync, which suffers from high communication latency. Further, in every superstep, as shown in Fig. 2 and Fig. 3, Sync is faster for PageRank and Async is faster for BFS-DO.

For BFS-LS and SSSP, as shown in Fig. 2 and Fig. 3, Sync benefits from message aggregation in the burst mode (i.e., the frontier is large), but it does not perform well compared to Async for all the graph workloads considered in this work. (See §III for testbed, benchmark algorithms and dataset details.)

D. Introducing the Hybrid Processing Design

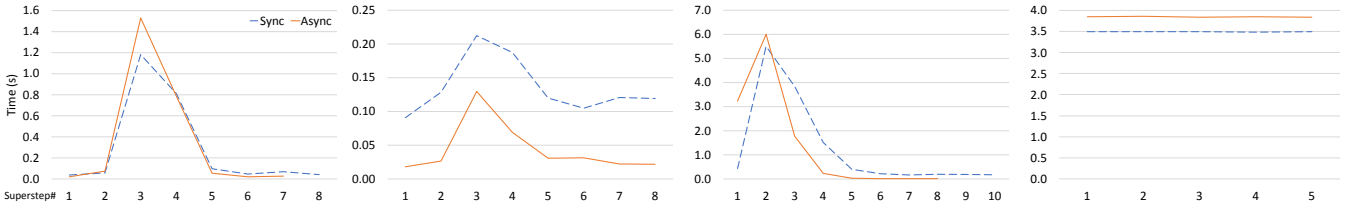
Based on the observation that, depending on the algorithm and the phase of its execution, Sync and Async modes hold advantages over one another, we explore a hybrid processing mode that combines Sync and Async processing within the same graph processing task: a superstep operates in either Sync or Async mode, and, over the course of execution, the system is able to switch between modes across supersteps.

The idea behind this design stems from two key observations (similar to those made by PowerSwitch yet in a shared-



(a) BFS-LS (b) BFS-DO (c) SSSP (d) PageRank

Fig. 2: Per superstep runtime comparison between Sync and Async designs, for BFS-LS (Level-Synchronous), BFS-DO (Direction-Optimized BFS), SSSP and Pagerank for **Twitter**. X-axis labels identify the supersteps and Y-axis shows runtime for each superstep.



(a) BFS-LS (b) BFS-DO (c) SSSP (d) PageRank

Fig. 3: Per superstep runtime comparison between Sync and Async designs, for BFS-LS (Level-Synchronous), BFS-DO (Direction-Optimized), SSSP and Pagerank for **RMAT-29**. X-axis labels identify the supersteps and Y-axis shows runtime for each superstep.

nothing platform context): Firstly, in the middle supersteps of a graph traversal algorithm, often the frontier size is significantly larger than in the rest of the execution (highlighted by prolonged runtime for these supersteps in Fig. 2 and Fig. 3). For Async, this also leads to a large volume of costly remote memory accesses (there is no message aggregation in Async). In this phase of the algorithm, Sync is a better choice: although it requires an explicit communication phase per superstep, Sync enables sequential remote-writes as well as message aggregation (which reduce the volume of remote accesses) - for a group of boundary edges with the same destination vertex, only one message is sent (per partition) over the interconnect. Secondly, Async is able to hide the overhead of remote accesses by overlapping communication with computation; thus, in the supersteps of an execution when the frontier size is relatively small the number of remote (random) accesses is low and Async is a better choice.

At the implementation level, the hybrid design retains the distributed state buffers used by Sync, i.e., per partition outbox and inbox. To support asynchronous processing, in the hybrid design, each partition also maintains pointers to the state buffers of all the other partitions.

Selecting a Processing Mode in the Hybrid Design. The decision to select either Sync or Async mode is informed by statistics about the current state of execution. The current implementation incorporates a low-overhead heuristic to accomplish this: if the global frontier size, i.e., the number of vertices, across all partitions, to be processed in the next superstep, is equal to or higher than a predefined threshold then Sync is selected, otherwise processing continues in the Async mode. For a traversal algorithm like BFS and SSSP, our observation is that the initial frontier size is always extremely small, therefore, for these algorithms, we always run the first superstep in Async mode. Running the first superstep in Async mode offers further benefits: it is able to activate remote

Algorithm 1 High-level Overview of the Hybrid Design

```
1: Input: graph G(V, E), threshold α
2: procedure RUN-ENGINE
3:     do
4:         if δ ≥ α then                          ▷ Online monitoring
5:             SUPERSTEP(mode←sync)
6:         else
7:             SUPERSTEP(mode←async)
8:         end if
9:     while not finished
10: end procedure
11: procedure SUPERSTEP
12:     if mode = sync then
13:         COMPUTE_SYNC();    barrier
14:         COMMUNICATE();     barrier
15:     else if mode = async then
16:         COMPUTE_ASYNC();   barrier
17:     end if                 ▷ COMPUTE invokes the algorithm kernel
18:     δ ← (f × 100)/|V|      ▷ f is the frontier size for the next superstep
19: end procedure
```

neighbors (i.e., they are included in the next frontier) as well, which is not possible in the Sync mode. Alg. 1 is a high-level overview of the key algorithmic steps in the hybrid design.

Determining the Threshold in the Hybrid Design. The threshold for the (minimum) frontier size (to switch to Sync mode) is provided as an input at application launch time. The thresholds used in this paper were determined through offline explorations[2], by studying a number of real-world graphs in the Sync and Async mode (presented earlier in this section and some of the results are in Fig. 2 and Fig. 3). Our observation is that the same threshold works for different graphs with similar properties, such as neighbor degree distribution and skewness. For example, for the power-law graphs, synthetic RMAT

[2]Past contributions have identified similar thresholds through offline studies of graph properties [9], [12].

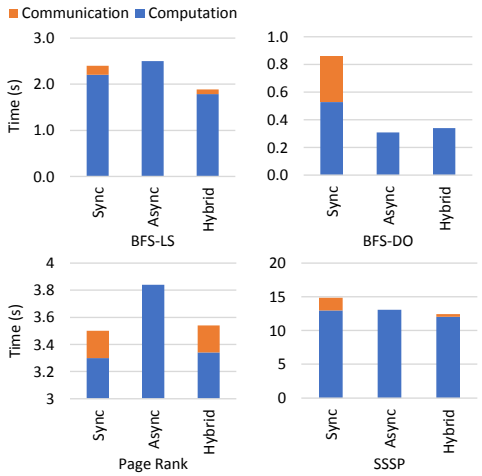


Fig. 4: Runtime comparison of three NUMA designs, for four algorithms, on a four socket configuration, using the RMAT-29 graph. Y-axis represents the runtime - lower is better. For Sync and hybrid designs, runtimes are broken down to computation and communication phases.

and real-world Twitter, the same threshold offers the best performance: when 30% (or more) of the vertices are present in the frontier, the Sync mode offers superior performance. We note that it might be possible to develop more informed heuristics for threshold selection (even an online heuristic), however, beyond the scope of our current contribution.

Comparison with PowerSwitch. PowerSwitch [9] extends GraphLab [1] and presents hybrid processing on a shared-nothing platforms. Unlike PowerSwitch that borrows the Sync and Async engines from GraphLab, we have designed both Sync and Async engines specifically for shared-memory (e.g., data layout for harnessing fast sequential read/write operations across NUMA domains). The switch-timing in PowerSwitch is epoch based: for Sync, few supersteps make an epoch; for Async, a fixed execution duration makes an epoch. To switch between modes, PowerSwitch relies on *user-provided parameters*, *offline profiling*, and *continuous online monitoring*. The required user-provided parameters are: convergence ratio, weight factor and sampling rate - determined through offline analysis by an expert on a per graph basis. PowerSwitch further leverages machine learning for predicting optimal switch-timing. In contrast, HyGN only takes the threshold as user input. At the end of each superstep, only the frontier size is verified against the threshold. Moreover, PowerSwitch requires separate state buffers for Sync and Async modes, and synchronization between them when switching between the processing modes.

E. Programming Interface / Algorithm Implementation

HyGN exposes a generic vertex-centric interface for algorithm implementation which is based on Totem [3], [13]. The COMPUTE_A/Sync function in Alg. 1 invokes the user provided algorithm kernel. For evaluation, we use four of the existing algorithms available with the latest Totem release. Our NUMA-aware graph engine is virtually a drop-in replacement for Totem's original CPU processing engine.

TABLE III: Graph datasets used for evaluation. Here, only RMAT graphs are synthetic, the rest are real-world graphs.

Graph	#Vertices	#Edges	Avg. Degree	Algorithms
Twitter (TWR)	52M	3.9B	48	All
Friendster (FRS)	66M	3.6B	55	All
webCC-2012 (WC12)	89M	4B	50	All
ukWeb-2007 (UK07)	105M	7.5B	71	All
clueWeb-2012 (CW12)	978M	85B	87	All
road-USA (RDUS)	24M	58M	2.4	SSSP
RMAT Scale 27–30, e.g., RMAT-30 (R30)	1B	32B	32	All

III. Evaluation

We study the performance of the three processing modes and empirically compare with five related projects. We demonstrate scalability on a four socket NUMA machine using real-world and synthetic graphs.

Testbed. The testbed is a Dell PowerEdge R920 with four Intel Xeon E7-4870 v2 (Ivy Bridge) sockets, each with 15 cores offering hyper-threading, and 30MB L3 cache; i.e., 60 cores or 120 hardware threads and 120MB L3 memory in total. The machine is equipped with 1.5TB main memory.

Benchmark Graph Algorithms. We use the following graph algorithms for evaluation; in the past, these algorithms have been commonly used by many [1]–[5], [7], [8], [14].

Level-Synchronous (**BFS-LS**) and Direction-Optimized (**BFS-DO**) are two variants of the Breadth-first Search (BFS) algorithm. For Scale-free graphs, BFS-LS suffers from heavy atomic writes in the middle supersteps (see supersteps 3 and 4 in Fig. 2(a) and Fig. 3(a)). BFS-DO addresses this issue by switching to the less expensive pull-based processing [12]. Single Source Shortest Path (**SSSP**) finds the shortest path from a source vertex to every other vertex in the graph; our implementation of SSSP is based on the Bellman-Ford algorithm. In **PageRank**, a vertex computes its rank based on ranks of its neighbors. Unlike BFS and SSSP, in PageRank, the compute workload (i.e., the frontier size) is the same across all iterations. We consider these algorithms for evaluation because they are the building blocks of many complex algorithms, and cover a spectrum of memory access patterns typically found in most graph algorithms. For example, BFS is used by Betweenness Centrality and Eccentricity algorithms. SPMV has similar memory access pattern as PageRank.

Datasets. Table III lists datasets used for evaluation. We create undirected versions of the graphs. The table also lists the algorithms used with the respective graphs for evaluation.

Twitter [15] and Friendster [16] are online social network graphs; webCC-2012 [16], ukWeb-2007 [17] and clueWeb-2012 [18] are real-world webgraphs; road-USA [16] is a road network graph with very large diameter (∼6K) and highly sparse (very low average degree). We use four synthetic Recursive MATrix (RMAT) graphs of scale 27–30 (the scale represents log of number of vertices - scale 29 has 2^{29} vertices). RMAT graphs are generated following the Graph500 standards with the parameters, (A, B, C, D) = (0.57, 0.19, 0.19, 0.05), and a directed edge factor of 16.

Experimental Methodology. We run each experiment 20 times and report the average. For BFS and SSSP, we use randomly selected source vertices (the same sources were used for all experiments when comparing frameworks). For PageRank, we run each experiment with 10 iterations and normalize the runtime to one iteration. For SSSP, we use edge weights in the range (0, 100] as used in [4]. We run BFS-DO using small diameter power-law graphs: Twitter, Friendster and RMAT; as noted by the authors, this algorithm primarily targets small diameter graphs [12].

We empirically compare our work with **Polymer** [7] - optimized for NUMA platforms; two NUMA-oblivious solutions: **Galois** v2.2 [4] and **Totem** [13] (past studies have shown Totem (CPU-only) outperforms Galois [13], and Galois generally performs better than the other well-known framework Ligra [7], [8].) Additionally, we compare with two distributed solutions: **HavoqGT** [2] - a fully asynchronous framework; and **PowerSwitch** [9] - it extends (and outperforms) GraphLab [1]. Since, except Galois, none of the above mentioned work offers switching between push and pull modes in the same execution, they do not implement BFS-DO.

Unless otherwise specified, an experiment was ran using all available 120 threads on our testbed.

A. Throughput and Runtime Performance

Fig. 5 compares the performance of the three NUMA designs. Here, to normalize across graphs of different sizes, the performance metric is Traversed Edges Per Second (TEPS) - a measure of throughput for a graph traversal algorithm by the Graph500 consortium (higher is better).

For PageRank, Sync performs up to 23% faster than Async as Async suffers from expensive remote random reads for remote neighbors, while the superior performance of Sync is attributed to local accesses only during computation and fewer remote accesses enabled by remote traffic aggregation. Since the frontier size never changes, Hybrid executes in Sync mode with little monitoring overhead (at most ∼14ms, ∼2% of the execution time).

Since BFS-DO alleviates the volume of memory writes in BFS-LS, Async benefits by doing a few remote random writes and less expensive random reads. Furthermore, as few updates happen, Sync suffers from expensive communication, as shown in Fig. 4. Async is up to 3.6× faster than Sync. Hybrid executes in Async mode only with overhead of up to 10% of execution time (∼32ms).

BFS-LS and SSSP have expensive burst modes, where Sync benefits from message aggregation. While Async benefits in rest of the supersteps by overlapping computation with communication. Note that in Async, remote random access happens for every edge, which becomes the bottleneck in the burst mode. For BFS-LS and SSSP, even though Sync is slower than Async (up to 2×), for most workloads, executing the burst phase in Sync mode improves the performance of Hybrid (from ∼7B TEPS to over 9B TEPS for BFS-LS, and over 2× speedup for SSSP). Fig. 6 shows that the hybrid mode improves the peak superstep runtime time compared to Async.

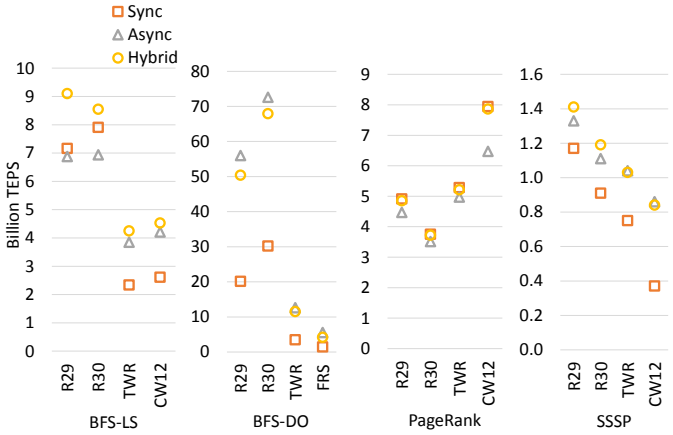


Fig. 5: Billion Traversed Edges Per Second (TEPS) achieved by the three NUMA designs, for four algorithms. Y-axis represents TEPS - higher is better.

When the frontier is large, the hybrid design takes advantage of remote traffic aggregation and fast sequential remote access (as it operates in the Sync mode), thus addresses the key limitation of the Async design.

B. Comparison with the Related Systems

Table IV presents runtime performance comparison of HyGN with five frameworks (using five datasets): NUMA-aware Polymer; NUMA-oblivious Galois and Totem; and distributed systems PowerSwitch (Sync+Async hybrid) and HavoqGT (Async). We run all experiments on all systems with 120 threads (120 MPI processes for HavoqGT) while Table V presents memory overhead..

Compared to NUMA-aware Polymer, HyGN is at most 14.2×, 3.3×, and 2.3× faster for BFS-LS, PageRank, and SSSP, respectively. HyGN always outperforms NUMA-oblivious and distributed solutions by a larger margin: HyGN is at most 45.8× faster than Galois and 2.2× faster than Totem. Among all the frameworks, PowerSwitch performs the worst: on average, HyGN is ∼31× faster (maximum ∼76×). The asynchronous distributed framework HavoqGT performs reasonably well on a single node, often outperforms Galois. On average HyGN is 8.6× faster than HavoqGT (maximum ∼20×). For SSSP, Polymer shows some advantage over our solution for social network graphs: ∼1.2× faster on average; but at the cost of consuming ∼5× more memory (Table V).

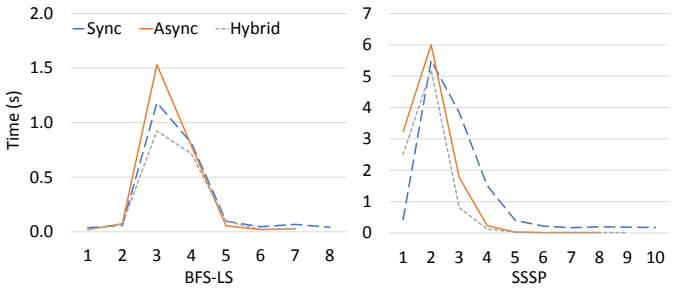


Fig. 6: Per superstep runtime comparison between Sync, Async and hybrid designs, for BFS-LS and SSSP using RMAT-29.

TABLE IV: Comparison of the runtime time (in seconds) between HyGN and five other frameworks. Since not all frameworks support BFS-DO, we only present numbers for BFS-LS. '×' indicates an unsuccessful run.

		NUMA-aware		NUMA-oblivious		Distributed	
	Graph	HyGN	Polymer	Totem	Galois	PowerSwitch	HavoqGT
BFS-LS	RMAT-27	**0.7**	3.4	0.9	5.8	42.7	8.9
	Twitter	**0.9**	12.8	1.5	3.1	33.4	7.5
	Friendster	**1.1**	5.8	1.8	5.2	18.1	8.9
	webCC12	**1.6**	12.4	2.5	6.3	82.9	10.1
	ukWeb07	**2.2**	19.1	4.0	8.1	×	12.3
PageRank	RMAT-27	**0.7**	2.3	0.9	2.0	18.8	10.4
	Twitter	**0.7**	1.9	1.1	13.9	13.3	8.3
	Friendster	**1.2**	2.2	1.9	2.0	91.7	9.9
	webCC12	**0.9**	1.8	2.7	41.2	13.8	9.6
	ukWeb07	**0.6**	0.9	0.8	1.8	×	12.4
SSSP	RMAT-27	2.4	**2.0**	5.3	13.3	65.7	11.2
	Twitter	3.3	**3.2**	6.6	8.7	62.4	35.5
	Friendster	8.9	**6.4**	15.1	21.6	45.1	11.2
	webCC12	**5.5**	6.7	11.8	19.9	137.7	13.0
	ukWeb07	**7.4**	16.7	11.3	31.6	×	18.6

PowerSwitch could not run ukWeb-2007 (and the other larger graphs) on a single node - the implementation has a hard limit on the length of the CSR edge list of a single partition (only one partition per compute node is supported), which the ukWeb-2007 graph exceeds. HyGN successfully ran the 85B edge clueWeb-2012 graph on our testbed (results in Fig. 5): the graph topology alone requires ∼640GB memory, while the weighted version requires about ∼1TB memory. Unfortunately, *none of the other frameworks, except for Totem, was able to run this graph*: Polymer fails with the out of memory (OOM) error, while Galois, PowerSwitch and HavoqGT exhausts the system memory during in-memory graph creation. (This is not unexpected of the distributed systems, since they have additional memory requirements to maintain states that enable efficient inter-node communication.)

To the best of our knowledge, two of the state-of-the-art shared-memory NUMA-aware graph processing frameworks are Polymer [7] and GraphGrind [8]. GraphGrind have shown to perform better than Polymer in some scenarios, but we were unable to compile their code which relies on a custom Cilk runtime, and the authors did not provide enough details to mitigate this issue. (We reached out to the authors for assistance; unfortunately, they were unable to help us to run their code.) Based on the numbers reported in [8], if we compare the relative improvements of GraphGrind over Polymer, on average, it is only 1.2× faster; and maximum up to 1.4× faster for the three algorithms (GraphGrind does not support BFS-DO). Our hybrid design is on an average 3.9× faster than Polymer.

Memory Consumption. Table V presents memory usages by three shared-memory frameworks for three algorithms and different graphs. Since it replicates application data and runtime states across partitions, on average, Polymer ends up using ∼5× more memory than HyGN. Memory consumption by Galois is on a par with HyGN, despite HyGN has the infrastructure overhead of distributed computation (and naturally requires more memory than Totem).

TABLE V: Memory consumption (in GB) by shared-memory frameworks. Lower is better.

Algorithm	Graph	HyGN	Polymer	Galois
BFS-LS	RMAT-27	18.5	122.0	**18.4**
	Twitter	**16.4**	93.2	17.3
PageRank	RMAT-27	**25.2**	123.0	32.7
	Twitter	**18.4**	93.8	49.3
SSSP	RMAT-27	**41.7**	174.0	47.9
	Twitter	**32.1**	140.0	40.8

C. Scalability Evaluation

Fig. 7 presents results of strong scaling: it plots runtimes on one, two and four sockets for HyGN, Polymer and Galois (when using RMAT-27 and Twitter graphs). With increasing number of sockets, NUMA-oblivious Galois shows poor, often negative, scaling: the speedup on four sockets over one socket, for Twitter, is maximum 1.7×; for RMAT-27 the average speedup is 1.4×, maximum 2.0×.

On four sockets HyGN shows respectable speedup over a single socket: for Twitter, the average is 3×, and the maximum is 3.9×. For RMAT-27, HyGN shows average 2.9× and maximum 3.7× speedup. Polymer does not show negative scaling; however, *on one socket it performs poorly, so the relative speedup on four sockets is not representative of true scalability (over 4× speedup)*. In [7], it was also shown that, on one socket, Polymer performs poorly and the slowest among all shared-memory frameworks.

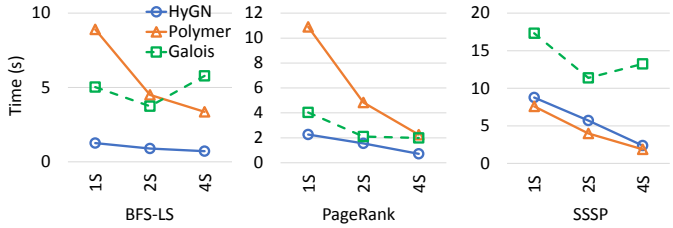
Fig. 8 compares workload scalability between HyGN and Polymer using RMAT graphs, where the graph size doubles with one fold increase in the RMAT scale. With increasing workload, for HyGN, the runtime grows more linearly compared to Polymer, whose performance worsen with the increasing graph size. For BFS-LS, HyGN is ∼4× faster for RMAT-27, which grows to ∼9× for RMAT-30. Polymer failed to run SSSP for the RMAT-30 graph; it crashes with the OOM error.

D. The Case of SSSP on the road-USA Graph

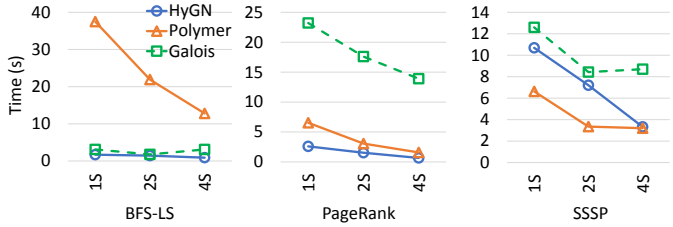
Table VI presents the number of supersteps required and runtime for SSSP for the road-USA graph for different designs. Furthermore, we study the impact of the threshold on the performance of the hybrid design.

Sync requires 70% more supersteps than Async: the road network graph has a large diameter and is highly sparse - in a superstep, the frontier size is extremely small; however, the need for an explicit communication phase severely slows down the convergence rate of the distributed states; over 80% of the total runtime time is spent in the communication phase. In the Async mode, the overhead of remote access are subsidized by overlapping theses accesses by compute cycles - the result is about an order of magnitude gain in runtime over Sync.

Table VI also shows runtimes for different thresholds: here, the system would operate in the Sync mode if the frontier has, for example, 0.1% of the vertices in the graph. Hybrid performance gradually improves with higher thresholds as



(a) RMAT-27



(b) Twitter

Fig. 7: Strong scaling experiment: Runtime for different numbers of CPU sockets for HyGN, Polymer and Galois, using RMAT-27 and Twitter graphs.

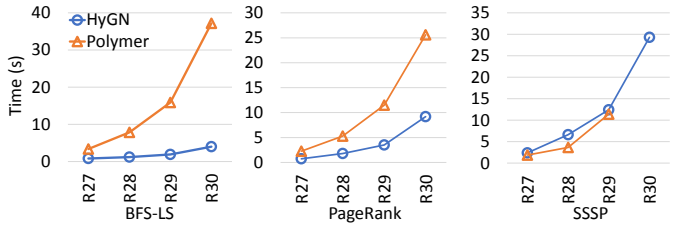


Fig. 8: Runtime when scaling the workload size for HyGN and Polymer using RMAT graphs. The experiment uses all sockets on the platform.

hybrid predominately operates in the Async mode (highlighted by similar runtime as Async-only mode).

IV. Related Work

To the best of our knowledge, Polymer [7] and GraphGrind [8] are state-of-the-art single-node NUMA-aware graph frameworks. Both frameworks extend Ligra [5] and adopt the vertex-cut partitioning approach (popular among distributed graph processing frameworks for scale-free, power-law graphs); only support the Sync processing mode; and store the data graph in a combined CSR-CSC format.

Polymer is memory inefficient as observed in our experiments (§III-B). It co-locates in- and out- edges with their source/target vertices and does vertex replication to reduce remote memory accesses. For application data, it avoids explicit messages for synchronization by having a single copy of them. GraphGrind adds a NUMA-aware extension to the Cilk runtime and addresses the load imbalance issue in graph processing on NUMA systems. From the memory usage perspective, it stores an additional copy of the graph for sparse traversal, and its memory requirement is comparable with that of Polymer [8]. In contrast, HyGN stores both graph topology and application specific data on the respective partition only. Hence, even in the Async mode, the interconnect traffic only accounts for updating state of the remote vertices, rather than updating the state of all vertices in a globally shared buffer as in Polymer and GraphGrind.

TABLE VI: Supersteps and execution time breakdown for SSSP on road-USA graph for Sync, Async, and Hybrid (for different thresholds).

Configuration	#Supersteps	Total time (s)	Compute time (s)	Comm. / Hybrid overhead (s)
Sync	575	34.49	5.96	28.53
Async	341	3.34	3.34	N/A
Hybrid (0.1%)	476	15.95	4.21	11.74
Hybrid (1%)	394	11.27	3.61	7.66
Hybrid (10%)	341	6.21	3.47	2.74

References

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[2] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates." in *SC*, 2014.

[3] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "On graphs, gpus, and blind dating: A workload to processor matchmaking quest." in *IPDPS*, 2013.

[4] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics." in *SOSP*, 2013.

[5] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory." in *PPoPP*, 2013.

[6] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *USENIX HOTOS*, 2015.

[7] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics." in *PPoPP*, 2015.

[8] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning." in *ICS*, 2017.

[9] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation." in *PPOPP*, 2015.

[10] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB Endow.*, 2014.

[11] H. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, and M. Snir, "A lightweight communication runtime for distributed graph analytics," in *IPDPS*, 2018.

[12] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search." in *SC*, 2012.

[13] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu, "Efficient large-scale graph processing on hybrid CPU and GPU systems," *Technical Report, CoRR*, 2013.

[14] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *in VLDB*, 2015.

[15] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *WWW*, 2010.

[16] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[17] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks." in *WWW*, 2011.

[18] T. L. Project, "The clueweb12 dataset," 2013. [Online]. Available: http://lemurproject.org/clueweb12